REMARKS

Claims 1-15 are all the claims pending in the application.

Claims 1-15 are rejected under 35 U.S.C. § 103 as being unpatentable over the cited prior art. In particular, the Examiner maintains the position that claims 1, 2, 7, 8 and 10-12 are unpatentable over Kwong (US 6,484,188) in view of Evans(US 6,836,884), claims 3, 4, and 13 are unpatentable over Kwong and Evans in further view of Benson (US 6,421,689), and claims 5, 6, 9, 14 and 15 are unpatentable over Kwong and Evans in further view of Traversat (US 6,854,115).

Rejection of Claims 1, 2, 7, 8 and 10-12

The Examiner maintains the rejection of claims 1, 2, 7, 8 and 10-12 under 35 U.S.C. § 103(a) as being unpatentable over Kwong in view of Evans. In response to our arguments presented in the Response filed on June 1, 2007, the Examiner asserts that col. 10, lines 5-25, of Evans discloses the claimed first and second memory units. In particular, the Examiner asserts that Evans discloses a run time system 100 which may load assembly 16 or portions thereof into memory for JIT compilations and execution via a class loader component 134. Applicants traverse the rejection as follows.

Applicants' invention relates to shortening the compiling time of byte codes in a Java program, wherein byte codes frequently used upon execution of the Java program are compiled and the resultant native codes are retrieved from memory and executed upon future execution of the Java program. Moreover, the native codes stored in memory are available for reuse in future executions of the program, even after the Java program has been completely executed.

Turning to the cited art, Kwong relates to the optimization of garbage collection code in the context of raw native interface function calls in the Java programming language. In particular, the method of Kwong selects a plurality of interface function calls to either eliminate or move within the program code by <u>analyzing either the byte code and the intermediate language</u>, or both, for either <u>inefficient code</u> constructs (i.e., unnecessary structures) or inefficient code containing native interface function calls to perform garbage collection (Abstract). In order to eliminate the unnecessary structures, a Java class file is input to a Java Classfile manager 201 which reads the class file information and sets everything up and produces a stream of byte code to be fed to a Java Bytecode Optimizer 202 (col. 6, lines 20-29). Thereafter, the Java Bytecode Optimizer 202 performs an analysis on the bytecode 100 in order to understand the bytecode, but does not actually change the bytecode 100 (col. 6, lines 30-36). Next, the bytecode is passed to the Bytecode to Native Interface Translator 203, which analyzes the bytecode stream for any interface calls and translates the bytecode stream into intermediate language to be passed to a Native Interface Specific Intermediate Language Optimizer 204 (col. 6, lines 49-65). The Native Interface Specific Intermediate Language Optimizer 204 analyzes interface calls of the intermediate language and determines what can be done to eliminate them or move them (e.g., an if-else statement) out of loops without changing the execution of the program (col. 7, lines 1-13). It is only after this optimization that the resultant code is passed to an Architecture Specific Code Generator 206 where the intermediate language is transformed into architecture native code 250 (col. 7, lines 45-51).

Evans, on the other hand, relates to a system for editing a software program in a common language runtime environment. In particular, Evans teaches suspending execution of the native code component at a first point, and allowing a user to edit the source code component to create

an edited source code component. The edited source code component is compiled using a source compiler to create an edited intermediate language component, and the edited intermediate language component is compiled using an intermediate language compiler to create an edited native code component. The edited native code component is then executed beginning at the point where execution was previously suspended. (Abstract).

The Examiner contends that Kwong in combination with Evans teaches each feature of claim 1. However, claim 1 recites:

> a class loader unit for loading byte codes generated by compiling Java program source codes;
> a first memory unit for maintaining the byte codes loaded by the class loader unit and native codes generated by compiling the byte codes in an accessible state;
> a second memory unit for storing the native codes that are loaded into the first memory unit in the accessible state;
> a native code manager unit for searching the native codes stored in the second memory unit and loading requested native codes into the first memory unit according to a request by a class loader unit; and
> an execution unit for executing the native codes that are loaded into the first memory unit in the accessible state.

The Examiner asserts that the Java Classfile Manager 201 of Kwong teaches the native code manager unit as claimed above. As stated above, however, the Java Classfile Manger 201 merely reads the Java class file information and sets everything up and produces a stream of byte code to be fed to a Java Bytecode Optimizer 202 for performing an analysis on the bytecode 100. According to Kwong, a native code has yet to be generated at this stage. Moreover, the Examiner concedes that Kwong fails to disclose the first memory unit and the second memory unit of claim 1. Thus, the Java Classfile Manager 201 (i.e., the alleged native code manager unit) is not operable to perform the claimed operations which require <u>searching native codes stored in the secondary memory unit and loading requested codes into the first memory unit</u>. Kwong fails

to teach or suggest such operations. Further still, the Java Classfile Manager 201 does not search among native codes stored in (second) memory to retrieve a native code of the native codes stored therein and loads the list into another (i.e., first) memory according to a request. At best, Kwong merely teaches a function call to the native code when execution is to be performed. However, no search as claimed is performed.

In addition, Kwong teaches when a native code is generated, all the needed interface calls, such as garbage collection function calls are inserted automatically (col. 5, lines 13-19). Thus, when an object (i.e., native code) is no longer in use, the Java run-time system automatically removes it from memory. Therefore, Kwong teaches against native codes stored in memory which are available for reuse in future executions of the program, even after the Java program has been completely executed.

As previously stated, the Examiner concedes that Kwong fails to disclose the first memory unit and the second memory unit of claim 1. Therefore, the Examiner cites to Evans to correct this deficiency. In particular, the Examiner asserts Evans teaches a run time system 100 which may load assembly 126 or portions thereof into memory for JIT compilations and execution via a class loader component 134 (col. 10, lines 5-25). The Examiner asserts that these features of Evans teach the claimed first memory unit and second memory unit of claim 1. Applicants respectfully disagree.

Evans teaches an intermediate language compiler such as a just-in-time (JIT) compiler 132, which is operable to compile intermediate language code (e.g., IL component 122) into native code (e.g., native code component 124) (col. 9, lines 32-36). An assembly 126 is created, which is an intermediate representation of the program (col. 9, lines 41-48). The assembly 126 comprises the <u>intermediate language component 122 and metadata 128</u> (col. 9, lines 45-46).

Thereafter, the runtime system 100 which may load assembly 126 or portions thereof into memory for JIT compilations and execution via a class loader component 134. However, assembly 126 does not contain any native code. Therefore, there is no native code being loaded or stored into memory as asserted by the Examiner. Instead, it is only after the above processes that the JIT compiler 132 compiles or converts the intermediate language code (e.g., from IL component 122) into corresponding native code (e.g., native code component 124). Evans, however, fails to teach or suggest a first memory unit for maintaining the byte codes loaded by the class loader unit and native codes generated by compiling the byte codes in an accessible state; and a second memory unit for storing the native codes that are loaded into the first memory unit in the accessible state. Furthermore, claim 1 requires the native code manager unit to operate in conjunction with the first and second memory units. Evans offers no teaching or suggestion of such features.

Moreover, Kwong teaches a method of optimizing code performance to increase runtime efficiency. Evans, on the other hand, teaches suspending execution of a program for editing errors in coding logic of a software program in a common language runtime environment. Therefore, Evans would actually slow down the execution of the program - the opposite purpose taught in Kwong. Moreover, edited code cannot be analyzed in Kwong. Kwong teaches that code is analyzed prior to the execution or runtime of the program. Evans, however, teaches editing code after the execution of a portion of the code. Assuming *arguendo* that edited code could be analyzed according to Kwong after the execution of a portion of the code and before execution of the program is resumed, this would entail performing the analysis on the bytecode each time the code is modified during runtime. This would result in slowing down execution times, rather than speeding up execution times (see Kwong, Abstract). Therefore, implementing

the teachings of Evans in Kwong would render Kwong inoperable for its intended purpose. Thus, there is no reason to combine the teaching of Evans with the teachings of Kwong as asserted by the Examiner.

In view of the above, Kwong, alone or in combination with Evans, does not teach or suggest all the features of claim 1. Thus, claim 1 should be patentable over the cited art.

Furthermore, Applicants submit that claim 8 is patentable for similar reasons and claims 2, 7 and 10-12 are patentable at least by virtue of their dependencies.

Rejection of Claims 3, 4 and 13

Claims 3, 4, and 13 stand rejected under 35 U.S.C. 103(a) as being unpatentable over Kwong in view of Evans in further view of Benson. However, Benson, alone or in combination with Kwong and Evans, does not correct the deficiencies of Kwong and Evans in regard to claims 1 and 8. Therefore, claims 3, 4 and 13 should be patentable at least by virtue of their dependencies.

Rejection of Claims 5, 6, 9, 14 and 15

Claims 5, 6, 9, and 14 have been rejected under 35 U.S.C. 103(a) as being unpatentable over Kwong in view of Evans and further view of Traversat. However, Traversat, alone or in combination with Kwong and Evans, does not correct the deficiencies of Kwong and Evans in regard to claims 1 and 8. Therefore, claims 5, 6, 9, 14 and 15 should be patentable at least by virtue of their dependencies.

Conclusion

In view of the above, reconsideration and allowance of this application are now believed to be in order, and such actions are hereby solicited. If any points remain in issue which the

Examiner feels may be best resolved through a personal or telephone interview, the Examiner is kindly requested to contact the undersigned at the telephone number listed below.

The USPTO is directed and authorized to charge all required fees, except for the Issue Fee and the Publication Fee, to Deposit Account No. 19-4880. Please also credit any overpayments to said Deposit Account.

Respectfully submitted,

Peter A. McKenna
Registration No. 38,551

SUGHRUE MION, PLLC
Telephone: (202) 293-7060
Facsimile: (202) 293-7860

WASHINGTON OFFICE
23373
CUSTOMER NUMBER

Date: October 29, 2007